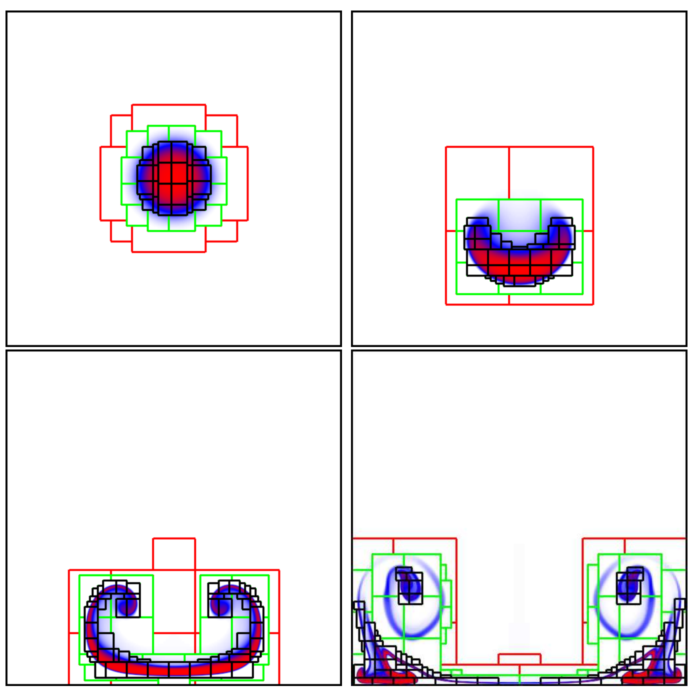
BoxLib User's Guide

J. Bell, A. Almgren, V. Beckner, M. Day, M. Lijewski
A. Nonaka, and W. Zhang

Center for Computational Sciences and Engineering
Lawrence Berkeley National Laboratory
https://ccse.lbl.gov

May 6, 2013



Contents

Preface

The current version of the BoxLib User's Guide can be found in the BoxLib git repository in BoxLib/docs. Visit our website at https://ccse.lbl.gov for free access to BoxLib. Any questions, comments, suggestions, etc., regarding this User's Guide should be directed to Andy Nonaka of CCSE at AJNonaka@lbl.gov. Further information about BoxLib can be found by contacting Ann Almgren of CCSE at ASAlmgren@lbl.gov or by visiting our website.

Chapter 1

Introduction

1.1 What is BoxLib?

BoxLib is a software library containing all the functionality to write massively parallel, block-structured adaptive mesh refinement (AMR) applications in two and three dimensions. BoxLib was developed at the Center for Computational Sciences and Engineering (CCSE) at Lawrence Berkeley National Laboratory and is freely available on our website at https://ccse.lbl.gov. The most current version of this User's Guide can be found in the BoxLib git repository at BoxLib/Docs. Any questions, comments, suggestions, etc., regarding this User's Guide should be directed to Andy Nonaka of CCSE at AJNonaka@lbl.gov. Further information about BoxLib can be found by contacting Ann Almgren of CCSE at ASAlmgren@lbl.gov or by visiting our website.

If you are new to BoxLib, we recommend you read Chapters 1 and 2 and familiarize yourself with the accompanying tutorial code. After working through Chapter 2, you will be able to run the tutorial code on as many cores as you like! Then, in Chapter 3 we enhance the Fortran90 tutorial code with additional features.

1.2 High-Level Overview

Key features of BoxLib include:

- C++/Fortran90 and pure Fortran90 versions
- Optional subcycling in time (C++ only)
- Support for cell-centered, face-centered, edge-centered, and nodal data
- Support for hyperbolic, parabolic, and elliptic solves on hierarchical grid structure
- Supports hybrid MPI/OpenMP parallel programming model
- Demonstrated scaling of linear solvers (parabolic and elliptic solvers) to 100,000 processors and hydrodynamics (hyperbolic solvers) to over 200,000 processors
- Plotfile format can be read by VisIt, yt, and AmrVis

- Basis of mature applications in combustion, astrophysics, cosmology, porous media, and fluctuating hydrodynamics
- Freely available on our website at `https://ccse.lbl.gov`

1.2.1 Parallel Programming Model

The fundamental parallel abstraction in `BoxLib` is the `MultiFab`, which holds the data on the union of grids at a level of refinement. A `MultiFab` is composed of multiple "Fortran array boxes" (i.e., `FArrayBox`es or `Fab`s); each `Fab` is a multidimensional array of data on a single grid. Whenever "work" needs to be done using data from a `MultiFab`, the `Fab`s composing that `MultiFab` are distributed among different processors to be worked on simultaneously. `Fab`s at each level of refinement are distributed independently. The software supports two data distribution schemes, as well as a dynamic switching scheme that decides which approach to use based on the number of grids at a level and the number of processors. The first scheme is based on a heuristic knapsack algorithm, which emphasizes load balancing; the second is based on the use of a Morton-ordering space-filling curve, which emphasizes on data locality for faster grid-to-grid communication. `MultiFab` operations are performed with an "owner computes" rule with each processor operating independently on its local data. For operations that require data owned by other processors, the `MultiFab` operations are preceded by a data exchange between processors to fill ghost cells. Each processor contains meta-data that is needed to fully specify the data locality and processor assignments of the `Fab`s. At a minimum, this requires the storage of an array of coordinates specifying the index space region for each box at each level of refinement. The meta-data can thus be used to dynamically evaluate the necessary communication patterns for sharing data amongst processors, enabling us to optimize communications patterns within the algorithm. By using a hybrid MPI-OpenMP approach to parallelization (see below), we are able to compute with fewer, larger grids, and thus the size of the meta-data is substantially reduced.

1.2.2 Hybrid MPI–OpenMP

The basic parallelization strategy uses a hierarchical programming approach for multicore architectures based on both MPI and OpenMP. In the pure-MPI instantiation, each `Fab` is assigned to a core, and each core communicates with every other core using only MPI. In the hybrid approach, where on each socket/node there are n cores that all access the same memory, we can divide our domain into fewer, larger grids, and assign each `Fab` to a socket/node, with the work associated with that grid distributed among the n cores using OpenMP.

1.2.3 Parallel I/O

Data for checkpoints and analysis are written in a self-describing format that consists of a directory for each time step written. Checkpoint directories contain all necessary data to restart the calculation from that time step. Plotfile directories contain data for post-processing, visualization, and analytics, which can be read using `VisIt`, `yt`, or `AmrVis` (a customized visualization package developed at CCSE for visualizing data on AMR grids, also freely available on our website). Within each checkpoint or plotfile directory is an ASCII header file and a subdirectory for each AMR level. The header describes the AMR hierarchy, including number of levels, the grids at each level, the problem size, refinement ratio between levels, time step, time, etc. Each of the subdirectories contains the data associated with the MultiFab for that level, which is stored in multiple files. Checkpoint and plotfile directories are written at user-specified intervals.

Restarting a calculation can present some difficult issues for reading data efficiently. In the worst case, all processors would need data from all files. If multiple processors try to read from the same file at the same time, performance problems can result, with extreme cases causing file system thrashing. Since the number of files is generally not equal to the number of processors and each processor may need data from multiple files, input during restart is coordinated to efficiently read the data. Each data file is only opened by one processor at a time. The IOProcessor creates a database for mapping files to processors, coordinates the read queues, and interleaves reading its own data. Each processor reads all data it needs from the file it currently has open. The code tries to maintain the number of input streams to be equal to the number of files at all times. Checkpoint and plotfiles are portable to machines with a different byte ordering and precision from the machine that wrote the files. Byte order and precision translations are done automatically, if required, when the data is read.

1.2.4 Scaling

In Figure 1.1 we present weak scaling results for several of our codes on the Cray XT5 Jaguarpf at OLCF. Jaguarpf has two hex-core sockets on each node. We assign one MPI process per node and spawn a single thread on each of the 12 cores. Results are shown for our compressible astrophysics code, CASTRO; the low Mach number code, MAESTRO; and our low Mach number combustion code, LMC. In the MAESTRO and CASTRO tests, we simulate a full spherical star on a 3D grid with one refined level (2 total levels). LMC is tested on a 3D methane flame with detailed chemistry using two refined levels. MAESTRO and LMC scale well to 50K-100K cores, whereas CASTRO scales well to over 200K cores. The overall scaling behavior for MAESTRO and LMC is not as close to ideal as that of CASTRO due to the communication-intensive linear solves performed at each time step. However, these low Mach number codes are able to take a much larger time step than explicit compressible formulations in the low Mach number regime.

1.3 BoxLib Directory Structure

BoxLib is the base directory in a hierarchy of subdirectories that support parallel, block-structured AMR applications in C++ and Fortran90. A schematic of the BoxLib directory structure is shown in Figure 1.2.

- Docs/

  Contains this BoxLib User's Guide.

- Src/

  BoxLib source code. The C++ source code is split into several directories. The Fortran90 source code is contained in one directory.

  - C_AMRLib/
  - C_BaseLib/

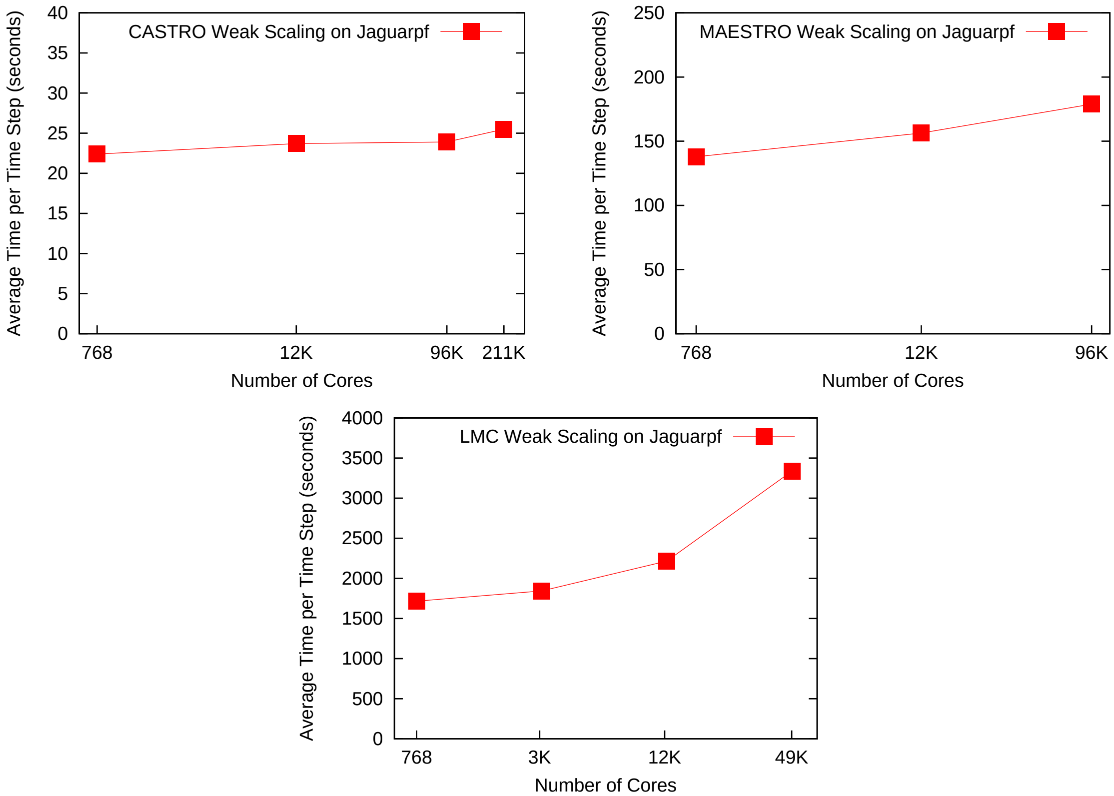


Figure 1.1: Weak scaling results for `CASTRO`, `MAESTRO`, and `LMC` on the Cray XT5 Jaguarpf at OLCF.

  - `C_BoundaryLib/`
  - `F_BaseLib/`
  - `LinearSolvers/`

    Source code for various linear solvers in C++ and Fortran90.

    * `C_CellMG/`
    * `C_NodalMG/`
    * `C_TensorMG/`
    * `C_to_F_MG/`
    * `F_MG/`

- `Tests/`

  Various tests used by `BoxLib` developers.

  - `C_BaseLib/`
  - `F_BaseLib/`
  - `LinearSolvers/`

- `Tools/`

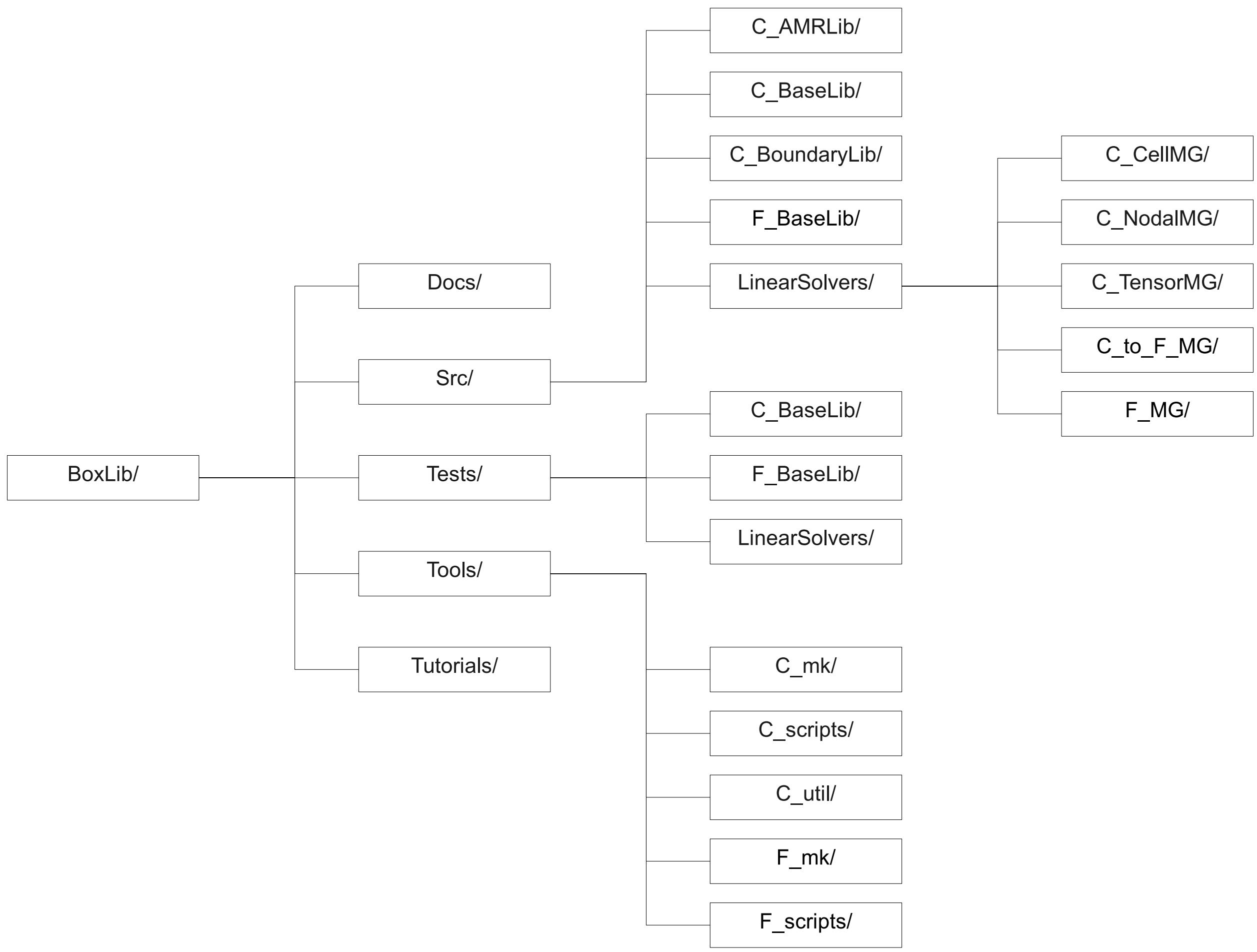


Figure 1.2: `BoxLib` directory structure.

- `C_mk/`
  The generic Makefiles that store the C++ compilation flags for various platforms.
- `C_scripts/`
  Some simple scripts that are useful for building, running, maintaining codes in C++.
- `C_Util/`
  Various utility codes for analyzing plotfiles.
- `F_mk/`
  The generic Makefiles that store the Fortran90 compilation flags for various platforms.
- `F_scripts/`
  Some simple scripts that are useful for building, running, maintaining codes in Fortran90.

- `Tutorials/`

  Contains sample codes referred to in this User's Guide.

Chapter 2

Getting Started

We now give an overview of common data structures used in BoxLib, followed by a simple example written in both Fortran90 and C++ that makes use of these structures. The example advances two scalar variables in time on a single level with multiple grids (no AMR) and produces plotfiles.

2.1 Overview of Data Structures

BoxLib contains the most fundamental objects used to construct parallel block-structured AMR applications. At each level of refinement, the region covered by that level is divided into grids, or boxes. The entire computational domain is covered by the coarsest (base) level of refinement (called level $\ell = 0$ in C++ and called level $\ell = 1$ in Fortran90) and can be represented on one grid or divided into many grids. Higher levels of refinement have cells that are finer by a "refinement ratio" of either 2 or 4 (in C++) or 2 (in Fortran90). The grids are properly nested in the sense that the union of grids at level $\ell + 1$ is contained in the union of grids at level ℓ. Furthermore, the containment is strict in the sense that, except at physical boundaries (i.e., domain boundaries that are not periodic), the level ℓ grids are large enough to guarantee that there is a border at least n_{buffer} (typically 4) level ℓ cells wide surrounding each level $\ell + 1$ grid (grids at all levels are allowed to extend to the physical boundaries so the proper nesting is not strict there). See Figure 2.1 for a sample two-dimensional grid structure.

On a grid, the data can be stored at cell-centers, faces, edges, or corners. In BoxLib, data that is on an face is termed 'nodal' in that one direction (see Figure 2.2). In three-dimensions (not pictured), data that is nodal in two directions is said to live on edges. Data that is nodal in all directions lives on the corners of cells (commonly referred to as the nodes). BoxLib uses 0-based spatial indexing, and for data that is nodal in one or more direction, the integer index corresponds to the lower boundary in that direction (see Figure 2.2). In our BoxLib applications, the state data (velocity, density, species, ...) is typically cell-centered. Fluxes are typically nodal in exactly one direction (i.e. they are face-centered). A few quantities are nodal in all directions (e.g. the pressure in the low Mach number projection methods).

- In C++ BoxLib, we must specify the number of spatial dimensions (1, 2, or 3), DIM, at compile-time. The code that will be built is specifically designed to run only with that number of dimensions.

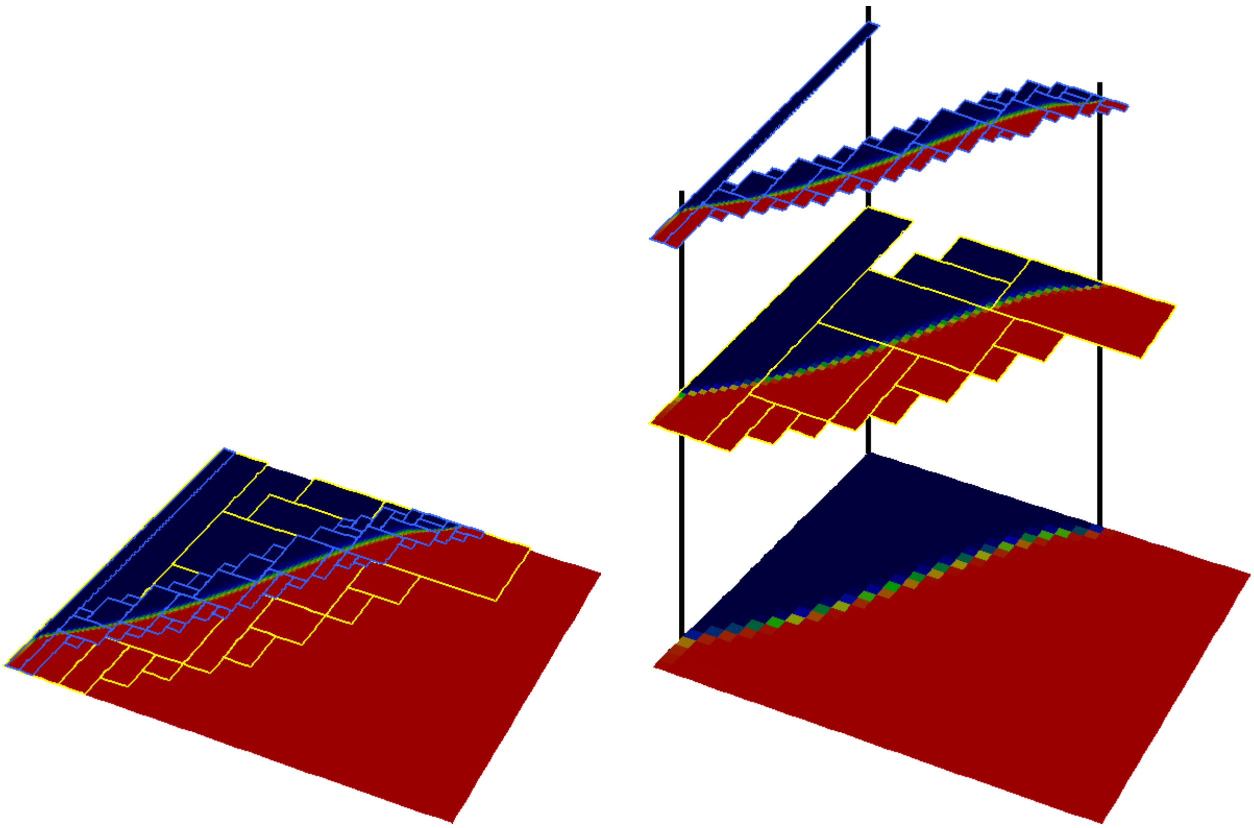


Figure 2.1: Sample grid structure with two levels of refinement. These grids satisfy the requirements that the base grid covers the entire computational domain and the grids are properly nested. Note that refined grids are allowed to extend to physical domain boundaries without coarser buffer cells.

- In Fortran90 `BoxLib`, we build dimension-independent code at compile-time, and tell the program the dimensionality of the problem via a runtime inputs file.

To simplify the description of the underlying AMR grid, `BoxLib` provides a number of classes. We now briefly summarize some of the major classes.

2.1.1 `IntVect`

`IntVect`s are `DIM`-tuples of integers that are used to define indices in space. In C++, an example of an `IntVect` in 2D would be (C++ source code will be shaded blue):

```
IntVect iv(3,5);
```

In Fortran90, we don't use `IntVect`s, but instead use standard arrays of integers (Fortran90 source code will be shaded green):

```
integer :: iv(2)
iv(1) = 3
iv(2) = 5
```

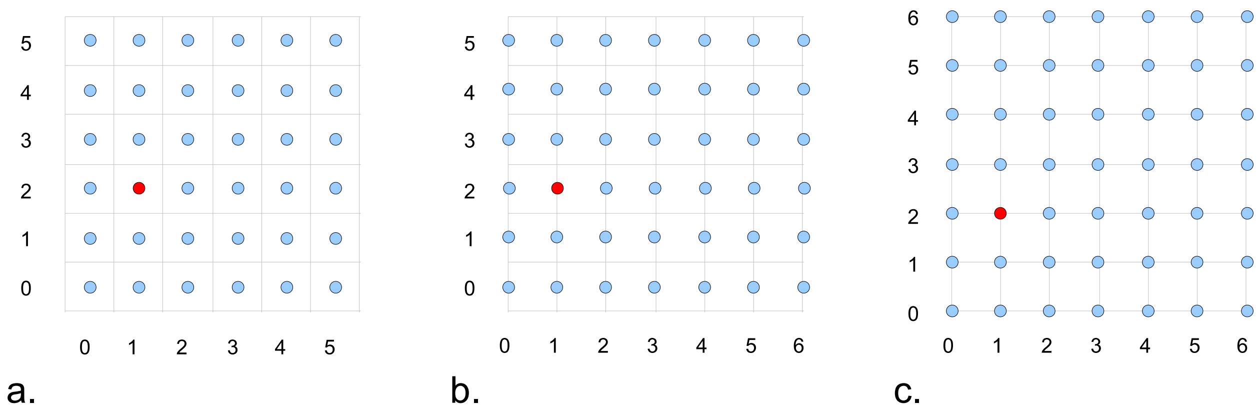


Figure 2.2: Some of the different data-centerings in two dimensions: (a) cell-centered, (b) nodal in the x-direction only (face-centered), and (c) nodal in both the x- and y-directions. Note that for data that is nodal in one or more direction, the integer index corresponds to the lower boundary in that direction. Also note that `BoxLib` uses 0-based indexing, e.g., in each of these centerings, the red point has the same indices: (1,2). Not shown is the case where data is nodal in the y-direction only. Also not shown is the three-dimensional edge-centered case, where the data is nodal in exactly two directions.

2.1.2 `Box`

A `Box` is simply a rectangular domain in space and does not hold any data. A `Box` contains the indices of its low end and high end, `IntVect lo` and `IntVect hi`.

- In C++, a `Box` also contains an `IndexType` (cell-centered, face-centered, or nodal) for each spatial direction.
- In Fortran90, a `Box` also contains the dimensionality of the `Box`.

To build a `Box` in C++ use:

```
IntVect iv_lo(0,0);
IntVect iv_hi(15,15);
Box bx(iv_lo,iv_hi);
```

To build a `Box` in Fortran90 use:

```
type(box) :: bx
integer   :: iv_lo(2), iv_hi(2)
iv_lo(1:2) = 0
iv_hi(1:2) = 15
bx = make_box(lo,hi)
```

The computational domain is divided into non-overlapping grids. The collection of grids at the same resolution comprise a level. Figure 2.3 shows three grids at the same level of refinement. Note that this figure cannot represent the base level of refinement, since it would require that the grids span the problem domain. The position of the grids is with respect to a global index space that covers the entire domain at that level and uses 0-based indexing. For example, the `Box` associated with grid 1 in the figure has `lo` = (2,6) and `hi` = (5,13).

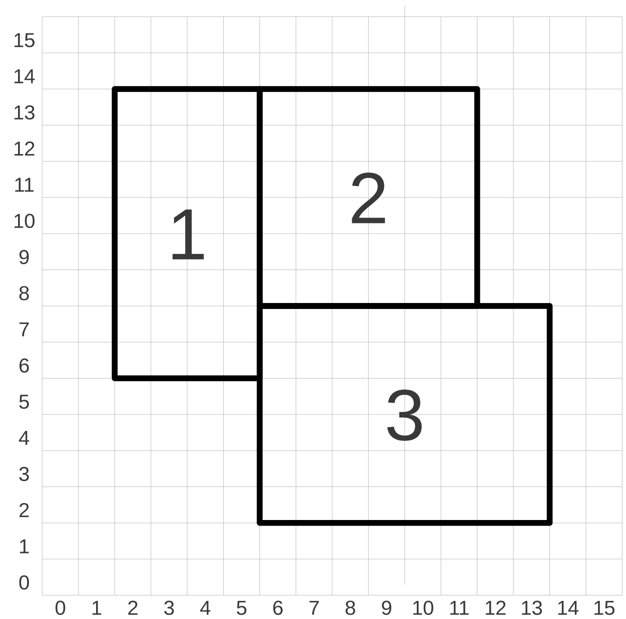


Figure 2.3: Three boxes that comprise a single level. At this level of refinement, the domain is 16×16 cells and the global index space runs from 0 to 15 in each coordinate direction. Note that these grids cannot be at the coarsest level, since it would require that the grids span the problem domain.

- Example: For a simulation with 32 cells in each direction at the coarsest level, the global index space for the coarsest level runs from 0 to 31 in each coordinate direction. Assuming refinement ratios of 2, the next finer level will have a global index space running from 0 to 63 in each coordinate direction (corresponding to 64×64 zones if fully refined), and the next finer level will have a global index space running from 0 to 127 in each coordinate direction (corresponding to 128×128 zones if fully refined).

2.1.3 `BoxArray`

A `BoxArray` is an array of `Boxes`. The size of the array is the number of `Boxes` in the `BoxArray`. Suppose your problem domain has lo indices (0,0) and hi indices (15,15), and you want to define a `BoxArray` to contain four 8×8 boxes to cover the problem domain. In Fortran90, you could do the following:

```
integer         :: lo(2), hi(2)
type(box)       :: bx(4)
type(boxarray) :: ba
lo(1) = 0
lo(2) = 0
hi(1) = 7
```

```
hi(2) = 7
bx(1) = make_box(lo,hi)
lo(1) = 8
lo(2) = 0
hi(1) = 15
hi(2) = 7
bx(2) = make_box(lo,hi)
lo(1) = 0
lo(2) = 8
hi(1) = 7
hi(2) = 15
bx(3) = make_box(lo,hi)
lo(1) = 8
lo(2) = 8
hi(1) = 15
hi(2) = 15
bx(4) = make_box(lo,hi)
call boxarray_build_v(ba,bx)
```

This is rather cumbersome, so instead we use other `BoxLib` functions to build the same `BoxArray`:

```
type(boxarray) :: ba
integer        :: lo(2), hi(2)
type(box)      :: bx
lo(1:2) = 0
hi(1:2) = 15
bx = make_box(lo,hi)
call boxarray_build_bx(ba,bx)  ! the boxarray has one 16^2 box
call boxarray_maxsize(ba,8)    ! the boxarray has four 8^2 boxes
```

The analogous code in C++ is

```
IntVect lo(0,0), hi(15,15);
Box bx(lo,hi);
BoxArray ba(bx);  // the BoxArray has one 16^2 box
ba.maxSize(8);    // the BoxArray has four 8^2 boxes
```

2.1.4 `layout` (Fortran90 Only)

A `layout` is a more intelligent `BoxArray`, since it contains a `BoxArray` as well as the associated processor assignments, `Box` connectivity, and many other parallel constructs. In the simplest case, if we have a `BoxArray` `ba` (obtained from the example above), a `layout` can be defined using:

```
type(layout) :: la
call layout_build_ba(la,ba)
```

In C++, the information that is contained in the Fortran90 `layout` is part of the `MultiFab` class.

2.1.5 `FArrayBox`

A `FArrayBox` (or `Fab`) is a "Fortran array box" that holds data. It contains the `Box` that it is built on as well as a pointer to the data that can be sent to a Fortran routine. In Fortran90, `Fab` data is stored in a four-dimensional array, `(nx,ny,nz,nc)` in size, regardless of the dimensionality of the problem. Here `nc` is the number of components, for instance representing different fluid variables. For 2D problems, `nz=1`.

In `BoxLib`, we don't usually deal with `Fab`s alone, but rather through `MultiFab`s, described next.

2.1.6 Floating point data

Floating point data in C++ is declared as `Real` which is `typedef` to either `float` or `double` depending on how `PRECISION` is set in the `GNUmakefile`. This is defined in `REAL.H`.

In Fortran90, the `bl_types` module defines a type `dp_t` that is double precision. Floating point types should be declared using `real (kind=dp_t)`.

2.2 The MultiFab

`MultiFab`s are so important that we will give them their own section. A `MultiFab` is a collection of all the `Fab`s at the same level of refinement.

- In C++, a `MultiFab` is defined using a `BoxArray`, number of components, and number of ghost cells that each `Fab` will have.
- In Fortran90, a `MultiFab` is defined using a `layout`, number of components, and number of ghost cells that each `Fab` will have.

A `MultiFab` has a "valid" region that is defined by the `BoxArray`or `layout`. Each `Fab` in the `MultiFab` is built large enough to hold valid data and ghost cell data, and thus the `Box` associated with each `Fab` is a grown version of the corresponding `Box` from the `BoxArray`. Thus, a `Fab` has no concept of ghost cells, it merely has a single `Box` that identifies it.

To build a `MultiFab`, we require a `layout` (in Fortran90) or a `BoxArray` (in C++). In Fortran90, using the `layout` built above, we build a `MultiFab` using:

```
type(multifab) :: data
call multifab_build(data,la,2,6) ! build the multifab with 2 components
                                 ! and 6 ghost cells

...                              ! do fun stuff with the data

call multifab_destroy(data)      ! free up memory to prevent leaks
```

In C++, using the `BoxArray` built above, you could either directly build a `MultiFab` using:

```
MultiFab data(ba,2,6);                    // build a MultiFab
```

or a pointer to a `MultiFab` using:

```
MultiFab* data = new MultiFab(ba,2,6);    // build pointer to MultiFab

...                                       // do fun stuff with the data
```

```
delete data;                                   // need to free the memory
```

2.2.1 Accessing `MultiFab` Data

Here is some sample Fortran90 code to access data within a `MultiFab`:

```
integer                    :: i,dm,ng,nc,lo(2),hi(2)
type(multifab)             :: data
real(kind=dp_t), pointer :: dp(:,:,:,:)

...     ! build multifab ‘‘data’’ as described above

dm = data%dim    ! dm is dimensionality
ng = data%ng     ! ng is number of ghost cells
nc = data%nc     ! nc is number of components

! loop over the grids owned by this processor
do i=1,nfabs(data)
   dp => dataptr(data,i)           ! dp points to data inside fab
   lo = lwb(get_box(data,i))       ! get lo indices of box
   hi = upb(get_box(data,i))       ! get hi indices of box
   select case(dm)
   case (2)
      call work_on_data_2d(dp(:,:,1,:), ng, nc, lo, hi)
   case (3)
      call work_on_data_3d(dp(:,:,:,:), ng, nc, lo, hi)
    end select
end do

! fill periodic domain boundary and neighboring grid ghost cells
call multifab_fill_boundary(data)

...

subroutine work_on_data_2d(data, ng, nc, lo, hi)

  integer           :: lo(2), hi(2), ng
  double precision :: data(lo(1)-ng:,lo(2)-ng:,:)

  ! local variables
  integer :: i,j,n

  do j=lo(2),hi(2)
     do i=lo(1),hi(1)
        do n=1,nc
           ! some silly function I made up
           data(i,j,n) = (i + j) * n
        end do
     end do
  end do
```

```
end subroutine work_on_data_2d
```

In C++:

```
#if    defined(BL_FORT_USE_UPPERCASE)
#define FORT_WORK_ON_DATA            WORK_ON_DATA
#elif  defined(BL_FORT_USE_LOWERCASE)
#define FORT_WORK_ON_DATA            work_on_data
#elif  defined(BL_FORT_USE_UNDERSCORE)
#define FORT_WORK_ON_DATA            work_on_data_
#endif

extern "C"
{
    void FORT_WORK_ON_DATA (
        Real* data, const int* Ncomp, const int* ng,
        const int* lo, const int* hi);
}

int main()
{
  int Ncomp = 2, Nghost = 6;

  ...    // build pointer to MultiFab* ``data'' as described above

  // MFIter is a ``MultiFab Iterator'' that essentially
  // loops over grids
  for ( MFIter mfi(*data); mfi.isValid(); ++mfi )
  {
    const Box& bx = mfi.validbox();
    FORT_WORK_ON_DATA((*data)[mfi].dataPtr(),
                      &Nghost, &Ncomp, bx.loVect(), bx.hiVect());
  }
  // fill periodic domain boundary and neighboring grid ghost cells
  data->FillBoundary();
}
```

The `FORT_WORK_ON_DATA` calls a Fortran90 subroutine which is nearly identical to the Fortran90 example given above. The only difference is the subroutine name cannot have the _2d or _3d in its name. Thus, the 2d and 3d versions are both named `subroutine work_on_data`, and must be written in different .f90 files, where the make system determines which version to compile based on `DIM`.

The `multifab_fill_boundary` and `FillBoundary` functions fill all ghost cells on periodic domain boundaries, as well as interior ghost cells with values that can simply be copied from the valid region of a neighboring grid at the same level of refinement. For single-level problems that are periodic in all directions, these functions fill all ghost cells. We will discuss non-periodic domain boundaries and fine grid ghost cells near coarse-fine interfaces in Chapter 3.1.

2.2.2 Other `MultiFab` Functions

`setVal` is a simple subroutine that sets the `MultiFab` data to a particular value. In Fortran90, use:

```
! set all variables to 0.0; ''all=.true.'' means set the ghost cells also
call setval(data,0.d0,all=.true.)
```

In C++, use:

```
data->setVal(0.0);  // set all variables to 0.0, including ghost cells
```

`copy` is a simple subroutine that copies data from one `MultiFab` to another. In Fortran90, use:

```
! copy components 1 and 2 from data_src into data_dest,
! including the ghost cells. calling sequence is
! (1) destination multifab, (2) first component of destination,
! (3) source multifab, ! (4) first component of source,
! (5) number of components, (6) ghost cells
call multifab_copy_c(data_dest,1,data_src,1,2,6)
```

In C++, use:

```
// copy components 0 and 1 from data_src into data_dest,
// including the ghost cells. calling sequence is
// (1) destination multifab, (2) source multifab,
// (3) first component of destination, (4) first component of source,
// (5) number of components, (6) ghost cells
MultiFab::Copy(*data_dest,*data_src,0,0,2,6)
```

There are many other subroutines available for adding, subtracting, multiplying, etc., components of `MultiFab`s, finding the min/max value, norms, number of cells, etc. Refer to `BoxLib/Src/F_BaseLib/multifab_f.f90` or `BoxLib/Src/C_BaseLib/MultiFab.H` for a complete listing.

2.3 Simple Example - Fortran90

We now provide a complete tutorial code that uses some concepts discussed above. The code also writes plotfiles that can be viewed, and can be run in parallel if you are working on a machine with MPI and/or OpenMP support. The Fortran90 version of this example is contained in `BoxLib/Tutorials/HeatEquation_EX1_F/`.

In this example, we advance the equation:

$$\frac{\partial \phi}{\partial t} = \nabla^2 \phi; \quad \phi(t=0) = 1 + e^{-100r^2}, \tag{2.1}$$

on a domain from [-1,1] in each spatial direction, where r is the distance to the point $(x,y,z) = (0.25, 0.25, 0.25)$. Note that we are placing the initial Gaussian profile slightly off-center. This asymmetry will be important in later sections when we examine the effects of non-periodic boundary conditions. We will assume that $\Delta x = \Delta y = \Delta z$ and use a fixed time step with $\Delta t = 0.9\Delta x^2$. We begin with a simple single-level, forward Euler discretization, periodic boundary conditions, and no refinement (i.e., only one level).

The basic time-advancement strategy uses the following temporal discretization:

$$\frac{\phi_{ij}^{n+1} - \phi_{ij}^{n}}{\Delta t} = \left[\nabla \cdot (\nabla \phi)\right]_{ij}. \tag{2.2}$$

In the explicit case, we first compute $\nabla\phi$, at faces using:

$$(\nabla \phi)_{i+1/2,j} = \frac{\phi_{i+1,j}^{n} - \phi_{ij}^{n}}{\Delta x}. \tag{2.3}$$

We will refer to these face-centered gradients as "fluxes". Next, we compute the update by taking the divergence of these fluxes,

$$\left[\nabla \cdot (\nabla \phi)\right]_{ij} = \frac{(\nabla\phi)_{i+1/2,j} - (\nabla\phi)_{i-1/2,j}}{\Delta x} + \frac{(\nabla\phi)_{i,j+1/2} - (\nabla\phi)_{i,j-1/2}}{\Delta y}. \tag{2.4}$$

We use a flux divergence formulation because it will enable a more natural extension to multiple levels of refinement, where we will be concerned with conservation across levels. Note that in this explicit case, since $\Delta x = \Delta y$, the Laplacian reduces to the standard five point stencil in two dimensions (seven point stencil in three dimensions).

Since the fluxes live on faces, we need face-centered `MultiFab`s, i.e., `MultiFab`s that are nodal in one spatial direction. In `advance.f90`, we build them as follows:

```
! an array of multifabs; one for each direction
type(multifab) :: flux(phi%dim)

! build the flux(:) multifabs
do i=1,dm
   ! flux(i) has 1 component, 0 ghost cells, and is nodal in direction i
   call multifab_build_edge(flux(i),phi%la,1,0,i)
end do
```

In the problem directory, you will see the following files:

- `GNUmakefile`

  This contains compiler settings and directories required by the make system to build the code.

  - `BOXLIB_HOME`
    Change this to point to the `BoxLib` home directory. Alternatively, you can define `BOXLIB_HOME` as an environment variable on your system.
  - `NDEBUG` ('`t`' or '`<blank>`') for TRUE or FALSE
    "not debug" (we know, confusing). If '`t`', modifies compiler flags to build a more optimized version of the code. The program will run faster, but have fewer runtime error checks.
  - `MPI` ('`t`' or '`<blank>`')
    Indicate whether you want your executable to be MPI-compatible. MPI must be installed on your machine in order to use this, and you must modify some of the make scripts, as will be discussed later.

– OMP ('t' or '<blank>')

  Turns on OpenMP compiler flags to compile in any OpenMP directives in the code. We will discuss OpenMP further in Section 2.7.

– PROF ('t' or '<blank>')

  Turns on timer compilation flags. Timers are useful for optimizing your code since they tell you what subroutines are taking the most time and require more optimization. Note that you still have to write timers into your code. We will discuss the implementation of timers in Section **??**.

– COMP ('gfortran, Intel, ...)'

  The Fortran compiler. Supported options include gfortran, Intel, PathScale, PGI, and Cray. The gfortran compiler seems to be bug-free on all systems we've run on, so stick with this unless you have good reason to change. Intel after version 9 seems flaky. PathScale (available at OLCF and NERSC) seems to work as long as you don't turn the optimization flags too high, and seems to run the fastest if you can actually get it to work. Cray seems to give similar performance as PathScale (perhaps because Cray bought out PathScale recently). PGI (available at OLCF and NERSC) seems to work fine, but is slower than the others.

– MKVERBOSE ('t' or '<blank>')

  Verbosity of compile-time output.

- GPackage.mak

  List of local files needed to be included in the build. The GNUmakefile points to this.

- main.f90, init_phi.f90, advance.f90, write_plotfile.f90

  Source code that is not within the BoxLib/Src/ tree. Note that if a file that exists in the BoxLib/Src/ tree also exists in the local directory, the local copy takes precedence as long as the GNUmakefile lists your local directory as a VPATH_LOCATIONS before the BoxLib source code directory, BoxLib/Src/F_BaseLib.

- inputs_2d, inputs_3d

  Input files to customize the simulation parameters.

To build the code, edit the GNUmakefile and simply type "make". An exectubale will appear that has some indication (but not complete) about what setting you used in the GNUmakefile. To run the code on a single processor, simply type, for example (terminal commands and non-source code files are shaded in red),

```
./main.Linux.gfortran.exe inputs_2d
```

The program will complete and there will be a series of plotfiles, e.g., plt00000, plt00100, etc., in the run directory. You can visualize the data and make animations using VisIt (available at https://wci.llnl.gov/codes/visit/); refer to Section 2.5.

2.4 Simple Example - C++

The C++ version of this example is contained in `BoxLib/Tutorials/HeatEquation_EX1_C/`.

- `GNUmakefile`

  This contains compiler settings and directories required by the make system to build the code.

  - `BOXLIB_HOME`

    Change this to point to the `BoxLib` home directory. Alternatively, you can define `BOXLIB_HOME` as an environment variables on your system.

  - `DEBUG` ('`TRUE`' or '`FALSE`')

    Debug mode. If 'FALSE', modifies compiler flags to build a more optimized version of the code. The program will run faster, but have fewer runtime error checks.

  - `USE_MPI` ('`TRUE`' or '`FALSE`')

    Indicate whether you want your executable to be MPI-compatible. MPI must be installed on your machine in order to use this, and you must modify some of the make scripts, as will be discussed later.

  - `USE_OMP` ('`TRUE`' or '`FALSE`')

    Turns on OpenMP compiler flags to compile in any OpenMP directives in the code. We will discuss OpenMP further in Chapter 2.7.

  - `PROFILE` ('`TRUE`' or '`FALSE`')

    Turns on timer compilation flags. Timers are useful for optimizing your code since they tell you what subroutines are taking the most time and require more optimization. Note that you still have to write timers into your code. We will discuss the implementation of timers in Chapter **??**.

  - `COMP` ('`g++`, `Intel`, ...)'

    The C++ compiler. Supported options include `g++`, `Intel`, `PathScale`, `PGI`, and `Cray`. See compiler notes above.

  - `FCOMP` ('`gfortran`, `Intel`, ...)'

    The Fortran compiler. See compiler notes above.

  - `DIM` ('1', '2', or '3')

    Dimensionality of the problem. Unlike Fortran90, you need to set this in the C++ version.

  - `PRECISION` ('`DOUBLE`' or '`FLOAT`')

    Precision of real numbers. You can use `FLOAT` for single-precision real numbers to save memory.

  - `EBASE` ('`main`', ...)

    The executable string will begin with this.

- `Make.package`

  List of local files needed to be included in the build. The `GNUmakefile` points to this.

- `main.f90`, `writePlotFile.cpp`, `writePlotFile.H`, `init_phi_2d.f90`, `init_phi_3d.f90`, `advance_2d.f90`, `advance_3d.f90`

  Source code that is not within the `BoxLib/Src/` tree. Note that if a file that exists in the `BoxLib/Src/` tree also exists in the local directory, the local copy takes precedence as long as the `GNUmakefile` lists your local directory in the `include` line before before the `BoxLib` source code directories.

- `inputs_2d`, `inputs_3d`

  Input files to customize the simulation parameters.

To build the code, simply type "`make`". An exectubale will appear that has some indication (but not complete) about what setting you used in the `GNUmakefile`. To run the code on one processor, simply type, for example,

```
./main2d.Linux.g++.gfortran.ex inputs_2d
```

The program will complete and there will be a series of plotfiles, e.g., `plt00000`, `plt00100`, etc., in the run directory. You can visualize the data and make animations using `VisIt` (available at `https://wci.llnl.gov/codes/visit/`); refer to Section 2.5.

2.5 Visualization Using `VisIt`

First, download and install `VisIt` from `https://wci.llnl.gov/codes/visit/`. To open a single plotfile, run `VisIt`, then select "File" → "Open file ...", then select the `Header` file associated the the plotfile of interest (e.g., `plt00000/Header`). Assuming you ran the simulation in 2D, here are instructions for making a simple plot:

- To view the data, select "Add" → "Pseudocolor" → "phi", and then select "Draw".
- To view the grid structure (not particularly interesting yet, but when we add AMR it will be), select " → "subset" → "levels". Then double-click the text "Subset - levels", enable the "Wireframe" option, select "Apply", select "Dismiss", and then select "Draw".
- To save the image, select "File" → "Set save options", then customize the image format to your liking, then click "Save".

Your image should look similar to the left side of Figure 2.4.

In 3D, you must apply the "Operators" → "Slicing" → "ThreeSlice", with the "ThreeSlice operator attribute" set to x=0.25, y=0.25, and z=0.25. You can left-click and drag over the image to rotate the image to generate something similar to right side of Figure 2.4.

To make a movie, you must first create a text file named `movie.visit` with a list of the `Header` files for the individual frames. This can most easily be done using the command:

```
~/BoxLib/Tutorials/HeatEquation_EX1_F> ls -1 plt*/Header | tee movie.visit
plt00000/Header
plt01000/Header
```

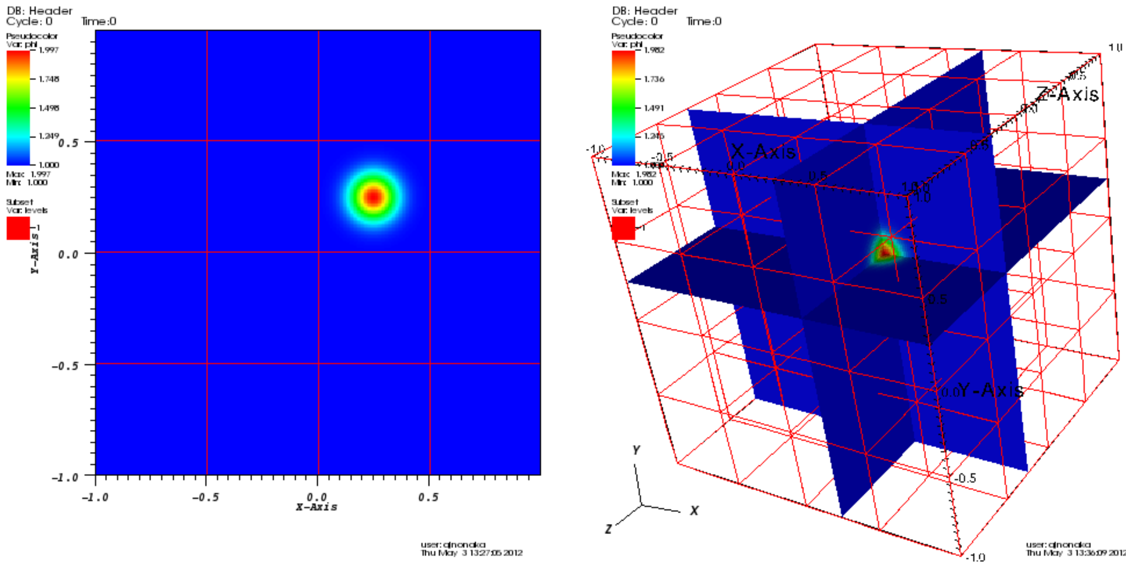


Figure 2.4: (Left) 2D image generated with VisIt. (Right) 3D image generated with VisIt.

```
plt02000/Header
plt03000/Header
plt04000/Header
plt05000/Header
plt06000/Header
plt07000/Header
plt08000/Header
plt09000/Header
plt10000/Header
```

The next step is to run `VisIt`, select "File" → "Open file ...", then select `movie.visit`. Create an image to your liking and press the "play" button on the VCR-like control panel to preview all the frames. To save the movie, choose "File" → "Save movie ...", and follow the on-screen instructions.

2.6 Running in Parallel with MPI

We will now demonstrate how to run the example in `BoxLib/Tutorials/HeatEquation_EX1_F/` using MPI parallelism. To run in parallel using C++ `BoxLib` is analogous. On your local machine, if you have MPI installed, you must first configure `BoxLib/Tools/F_mk/GMakeMPI.mak` and `BoxLib/Tools/C_mk/Make.mpi` to your MPI installation. Then, you can simply build the executable as describe before, but with `MPI=t` in the `GNUmakefile`. Alternatively, you can override the settings in `GNUmakefile` at the command line using, e.g., "`make MPI=t`". An executable named `main.Linux.gfortran.mpi.exe` will be built. Then you can run the program in parallel using, e.g.,

```
mpiexec -n 4 main.Linux.gfortran.mpi.ex inputs_2d
```

To run in parallel on the hopper machine at NERSC, first copy the BoxLib source code into your home directory on hopper and go to the HeatEquation_EX1_F/ directory. The default programming environment uses the PGI compilers, so we will switch to the gnu programming environment to make g++ and gfortran available using the command:

```
module swap PrgEnv-pgi PrgEnv-gnu
```

Next, in GNUmakefile, set MPI=t, and then type "make" (or alternatively, type "make MPI=t"). An executable named main2d.Linux.g++.gfortran.mpi.exe will be built. You cannot submit jobs in your home directory, so change to a scratch space ("cd $SCRATCH" will typically do), and copy the executable and inputs_2d into this directory. Then you need to create a job script, e.g., "hopper.run", that has contents (for tcsh):

```
#PBS -q debug
#PBS -l mppwidth=48
#PBS -l walltime=00:05:00
#PBS -j eo

cd $PBS_O_WORKDIR

echo Starting `date`

aprun -n 48 ./main2d.Linux.gfortran.mpi.ex inputs_2d

echo Ending `date`
```

Note that "mppwidth" and "-n" both indicate the number of cores you are requesting. To run, simply type "qsub hopper.run". You can monitor the status of your job using "qstat -u <username>" and view your position in the queue using "showq".

2.7 Running in Parallel with MPI/OpenMP (3D ONLY)

Both the C++ and Fortran versions of this tutorial also support hybrid MPI/OpenMP parallelism for **three-dimensional problems only**. You may add OMP parallelism to two-dimensional work loops, but be advised that subroutines within the BoxLib infrastructure are not threaded for two-dimensional problems. To "thread" the code, we have simply added OpenMP directives (using the !$omp parallel do construct) to any i/j/k loops we were interested in threading. For example, in init_phi.f90:

```
!$omp parallel do private(i,j,k,x,y,z,r2)
do k=lo(3),hi(3)
   z = prob_lo(3) + (dble(k)+0.5d0) * dx
   do j=lo(2),hi(2)
      y = prob_lo(2) + (dble(j)+0.5d0) * dx
      do i=lo(1),hi(1)
         x = prob_lo(1) + (dble(i)+0.5d0) * dx

         r2 = ((x-0.25d0)**2 + (y-0.25d0)**2 + (z-0.25d0)**2) / 0.01d0
         phi(i,j,k) = 1.d0 + exp(-r2)
```

```
      end do
   end do
end do
!$omp end parallel do
```

This User's Guide is not a manual on OpenMP, so simply note that this particular construct tells each thread to work on different values of `k`, with each thread getting its own local copy of `i`, `j`, `x`, `y`, `z`, and `r2`.

Finally, to tell the compiler that we would like to run with OpenMP, we make sure to set `OMP=t` (in Fortran) or `USE_OMP=TURE` (in C++) in the `GNUmakefile`. Otherwise, all OpenMP directive are simply ignored. Note that at runtime you must have set the `OMP_NUM_THREADS` environment variable properly in order for threads to work. Also, note that you can enable/disable MPI independently from the OMP flag. Finally, here is a sample hopper script (tcsh) for a hybrid MPI/OpenMP job:

```
#PBS -q debug
#PBS -l mppwidth=48
#PBS -l walltime=00:05:00
#PBS -j eo

setenv OMP_NUM_THREADS 6

cd $PBS_O_WORKDIR

echo Starting `date`

aprun -n 8 -N 4 -S 1 -ss -d 6 ./main.Linux.gfortran.mpi.omp.exe inputs_2d

echo Ending `date`
```

- "mppwidth": how many total cores requested
- "-n": total number of MPI tasks
- "-N": number of MPI tasks per hopper node
- "-S": number of MPI tasks per NUMA node
- "-ss": demands strict memory containment per NUMA node
- "-d": number of OpenMP threads per MPI task

Chapter 3

Advanced Topics With Fortran90 BoxLib

We now enhance our heat equation example from the previous section. Below is an outline of how we will proceed. Each of these sections contains an accompanying tutorial code that builds upon the previous example.

- In Section 3.1 we develop the capability to handle other (non-periodic) boundary condition types.
- In Section 3.2 we develop the capability to have multiple levels of refinement using a fixed, multilevel grid structure.
- In Section 3.3 we develop the capability to adaptively change the multilevel grid structure.
- In Section 3.4 we develop the capability to solve the equation implicitly, using the linear solver libraries.

3.1 Boundary Conditions

In order to understand how to implement boundary conditions, we shall first describe the general principles behind working with boundary conditions. The `BoxLib/Tutorials/HeatEquation_EX2_F/` tutorial continues our heat equation example, but now with some non-periodic boundary condition support. The boundary condition modules in `BoxLib/Src/F_BaseLib/define_bc_tower.f90` and `multifab_physbc.f90` can be used as a springboard for developing your own customized boundary conditions.

3.1.1 General Principles

The basic idea is that every grid has knowledge of the boundary condition type at the low and high side edge in each direction. The "physical" boundary condition types supported by default are `INLET`, `OUTLET`, `SYMMETRY`, `SLIP_WALL`, `NO_SLIP_WALL`, and `PERIODIC`. There is also an `INTERIOR` boundary condition type, which will be explained below. We use an integer mapping that is contained in `BoxLib/Src/F_BaseLib/bc.f90`:

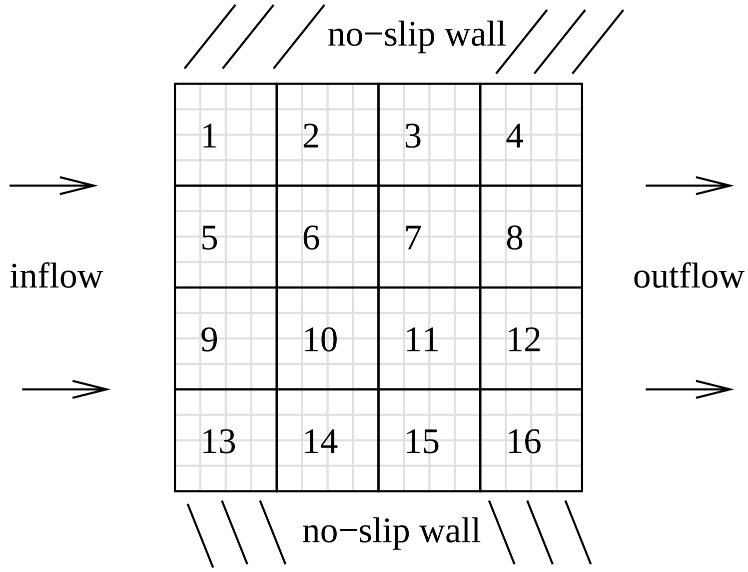


Figure 3.1: Two-dimensional example with 16 - 4^2grids with INLET, OUTLET, and NO_SLIP_WALL boundary conditions. The numbers refer to the grid number.

```
integer, parameter, public :: PERIODIC     = -1
integer, parameter, public :: INTERIOR     =  0

integer, parameter, public :: INLET        = 11
integer, parameter, public :: OUTLET       = 12
integer, parameter, public :: SYMMETRY     = 13
integer, parameter, public ::    SLIP_WALL = 14
integer, parameter, public :: NO_SLIP_WALL = 15
```

Examples:

- Consider grid 1 in Figure 3.1. The low-x boundary condition is INLET, and the high-y boundary condition is NO_SLIP_WALL. The high-x and low-y boundary conditions are INTERIOR, which means that the ghost cells share the same physical space as cells in the valid region of another grid. Note that for grids 6, 7, 10, and 11, the boundary condition type for every side is INTERIOR.
- Figure 3.2 demonstrates a problem with periodicity in the x-direction. In this case, the low-x boundary condition for grid 1 is PERIODIC. Note there are some similarities between PERIODIC and INTERIOR boundary conditions when it comes to filling ghost cells in that ghost cell values are simply copied in from the valid region of another grid. In fact, one can think of PERIODIC as just a special type of INTERIOR boundary condition. For the other boundary conditions types, the user can write custom boundary conditions routines to fill ghost cells, which can involve setting ghost cell values directly, or using interior points and/or physical boundary conditions in some stencil operation.
- Now, consider an example with refined grids. Figure 3.3 contains three grids at the next level of refinement. In this case, for grid 1, all of the boundary condition types are INTERIOR, even

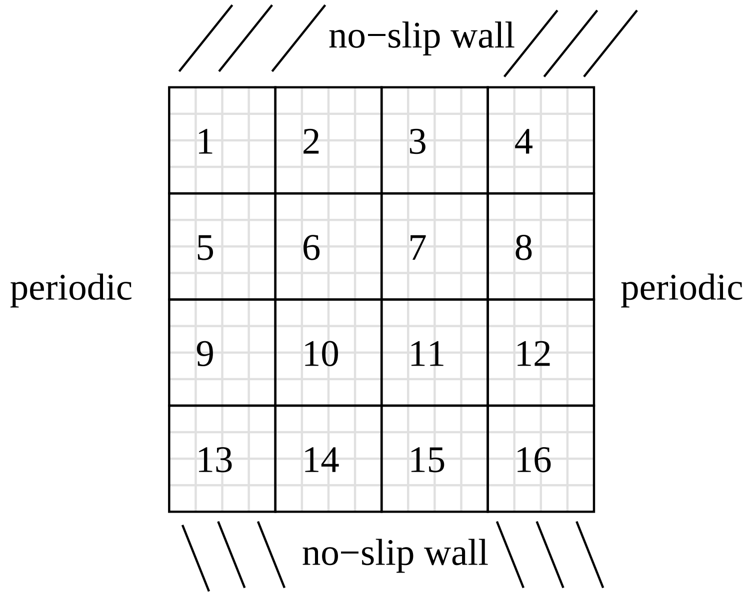


Figure 3.2: Two-dimensional example with 16 - 4^2grids with `PERIODIC` and `NO_SLIP_WALL` boundary conditions. The numbers refer to the grid number.

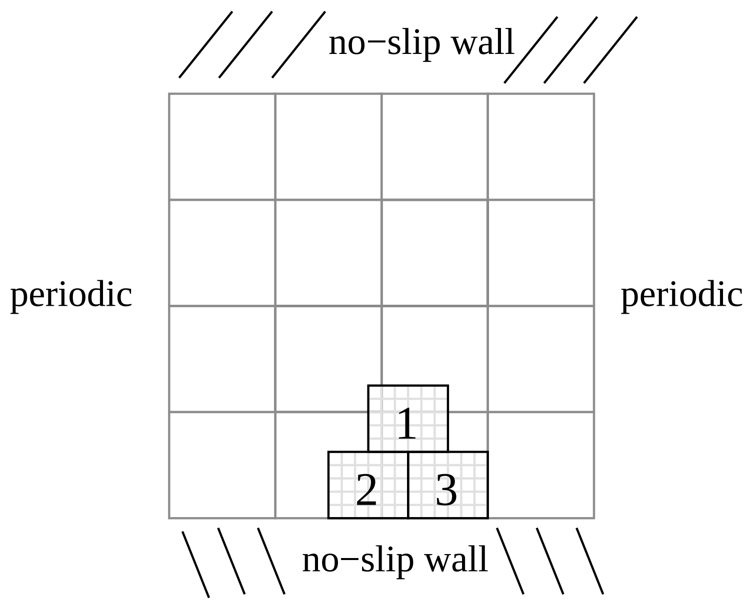


Figure 3.3: Two-dimensional example with 3 grids at a finer resolution than the base grid.

though the neighboring valid region data is at a coarser level of refinement. For grid 2, the low-y boundary condition is `NO_SLIP_WALL`, and the other three walls are `INTERIOR`.

3.1.2 Implementation

Typically, we read in integer values from the inputs file for `bc_x_lo`, `bc_x_hi`, `bc_y_lo`, etc., that correspond to the physical boundary condition types. We then build a `bc_tower` object which is an array of `bc_level` objects, one for each level of refinement. The `bc_level` contains several integer array data structures, as can be seen in `BoxLib/Src/F_BaseLib/define_bc_tower.f90`:

```
type bc_level

   ! 1st index is the grid number (grid "0" corresponds to the prob domain)
   ! 2nd index is the direction (1=x, 2=y, 3=z)
   ! 3rd index is the side (1=lo, 2=hi)
   ! 4th index is the variable (only assuming 1 variable here)
   integer, pointer :: phys_bc_level_array(:,:,:) => Null()
   integer, pointer ::  adv_bc_level_array(:,:,:,:) => Null()
   integer, pointer ::  ell_bc_level_array(:,:,:,:) => Null()

end type bc_level
```

Each level has a `phys_bc_level_array(0:ngrids,dim,2)` array, where `ngrids` is the number of grids on that level, `dim` is the dimensionality of the simulation, and the third index refers to the lower or upper edge of the grid in that coordinate direction. This stores the *physical description* of the boundary type (`INLET`, `OUTLET`, etc.), which is independent of the variables that live on the grid. The `phys_bc_level_array(0,:,:)` refers to the entire domain. If an edge of a grid is not a physical boundary, then it is set to a default value, typically `INTERIOR`. These boundary condition types are used to interpret the actual method to fill the ghost cells for each variable, as described in `adv_bc_level_array` and `ell_bc_level_array`.

Whereas `phys_bc_level_array` provides a physical description of the type of boundary, the array `adv_bc_level_array` describes the action to be taken (e.g. reflect, extrapolate, etc.) for each variable when filling physical ghost cells on domain boundaries. The prefix "`adv_`" is somewhat of a misnomer, as this data structure was originally intended to tell advection (or hyperbolic) solvers how to fill ghost cells, but now is generally used to fill physical domain boundary ghost cells in any instance where the user needs to set them. The form of this array is `adv_bc_level_array(0:ngrids,dim,2,nvar)` where the additional component, nvar, allows for different state variable that lives on a grid to have different boundary condition actions associated with it. For example, you could have `nvar=1` correspond to the x-velocity, `nvar=2` correspond to density, and `nvar=3` correspond to pressure. In the `BoxLib/Tutorials/HeatEquation_EX2_F/` tutorial, there is only one variable, ϕ, so obviously `nvar=1` shall correspond to ϕ. When we build the `adv_bc_level_array`, we first set all values to `INTERIOR`, and then overwrite any physical domain boundary condition types, as given in `phys_bc_level_array`. The `adv_bc_level_array` types supported by default are (as listed in `BoxLib/Src/F_BaseLib/bc.f90`):

```
integer, parameter, public :: INTERIOR     =  0

integer, parameter, public :: REFLECT_ODD  =  20
integer, parameter, public :: REFLECT_EVEN =  21
integer, parameter, public :: FOEXTRAP     =  22
integer, parameter, public :: EXT_DIR      =  23
integer, parameter, public :: HOEXTRAP     =  24
```

To manually fill ghost cells, we call `multifab_physbc`, passing in the state multifab along with the `adv_bc_level_array`. The subroutines `physbc_1d/2d/3d` in `BoxLib/Src/F_BaseLib/multifab_physbc.f90`, indicate how to fill ghost cells. For example,

```
  subroutine multifab_physbc(s,start_scomp,start_bccomp,ncomp, &
                             the_bc_level,time_in,dx_in, &
                             prob_lo_in,prob_hi_in)

    integer        , intent(in   )           :: start_scomp,start_bccomp
    integer        , intent(in   )           :: ncomp
    type(multifab) , intent(inout)           :: s
    type(bc_level) , intent(in   )           :: the_bc_level
    real(kind=dp_t), intent(in   ), optional :: time_in,dx_in(:)
    real(kind=dp_t), intent(in   ), optional :: prob_lo_in(:),prob_hi_in(:)

    ! Local
    integer                  :: lo(get_dim(s)),hi(get_dim(s))
    integer                  :: i,ng,dm,scomp,bccomp
    real(kind=dp_t)          :: time,dx(get_dim(s))
    real(kind=dp_t)          :: prob_lo(get_dim(s)),prob_hi(get_dim(s))
    real(kind=dp_t), pointer :: sp(:,:,:,:)

    ! set optional arguments
    time    = 0.d0
    dx      = 0.d0
    prob_lo = 0.d0
    prob_hi = 0.d0
    if (present(time_in))       time = time_in
    if (present(dx_in))           dx = dx_in
    if (present(prob_lo_in)) prob_lo = prob_lo_in
    if (present(prob_hi_in)) prob_hi = prob_hi_in

    ng = nghost(s)
    dm = get_dim(s)

    do i=1,nfabs(s)
       sp => dataptr(s,i)
       lo = lwb(get_box(s,i))
       hi = upb(get_box(s,i))
       select case (dm)
       case (2)
          do scomp=start_scomp,start_scomp+ncomp-1
             bccomp = start_bccomp + scomp - start_scomp
             call physbc_2d(sp(:,:,1,scomp), lo, hi, ng, &
                          the_bc_level%adv_bc_level_array(i,:,:,bccomp), &
                          time, dx, prob_lo, prob_hi)
          end do
...

  subroutine physbc_2d(s,lo,hi,ng,bc,time,dx,prob_lo,prob_hi)

    use bl_constants_module
```

```
    use bc_module

    integer         , intent(in   ) :: lo(:),hi(:),ng
    real(kind=dp_t), intent(inout) :: s(lo(1)-ng:,lo(2)-ng:)
    integer         , intent(in   ) :: bc(:,:)
    real(kind=dp_t), intent(in   ) :: time,dx(:),prob_lo(:),prob_hi(:)

    ! Local variables
    integer :: i,j

    !!!!!!!!!!!!!
    ! LO-X SIDE
    !!!!!!!!!!!!!

    if (bc(1,1) .eq. EXT_DIR) then
       ! set all ghost cell values to a prescribed dirichlet
       ! value; in this example, we have chosen 1
       do j = lo(2)-ng, hi(2)+ng
          s(lo(1)-ng:lo(1)-1,j) = 1.d0
       end do
    else if (bc(1,1) .eq. FOEXTRAP) then
       ! set all ghost cell values to first interior value
       do j = lo(2)-ng, hi(2)+ng
          s(lo(1)-ng:lo(1)-1,j) = s(lo(1),j)
       end do
...
```

Note that the optional arguments allow for the use of space and/or time-dependent boundary conditions.

`ell_bc_level_array` is the analog to `adv_bc_level_array` for the linear solvers in `BoxLib`. These will be described in Section 3.4.

3.2 Multiple Levels of Refinement

In the `BoxLib/Tutorials/HeatEquation_EX3_F/` tutorial, we have expanded our example to the cases of multiple levels of refinement, with the grids fixed in space. Note that there is currently no subcycling support for Fortran90 `BoxLib`, so in this example we advance all the grids with the same time step, and perform synchronization operations between levels.

The big change for this tutorial is that we use a "multilevel layout" `ml_layout` rather than a `layout`, and also `multifab phi` and `dx` are now `nlevs` sized arrays. After initializing or updating ϕ, we must fill all ghost cell and synchronize the solution between levels. After we make the fluxes, we must synchronize the fluxes to maintain conservation.

There are three key subroutines for filling ghost cells and synchronizing data in multilevel applications. Each of these involves a coarse level and a fine level:

- `ml_cc_restriction` sets coarse cell-centered values equal to the average of the fine cells covering it.

- `ml_edge_restriction` sets coarse edge-centered values (such as fluxes) equal to the average of the fine edges covering it.
- `multifab_fill_ghost_cells` fills fine ghost cells using interpolation from the underlying coarse data. Note that this operation does not affect ghost cells that would be filled by `multifab_fill_boundary` or `multifab_physbc`.

3.3 Adaptive Mesh Refinement

Now fully implemented in `BoxLib/Tutorials/HeatEquation_EX4_F/`. The basic idea is to "tag" the cells you with to refine in `BoxLib/Src/F_BaseLim/tag_boxes.f90`. To write your own customized tagging criteria, copy `tag_boxes.f90` into your local directory and modify it, since this copy will take precedence over the version in the `BoxLib` source.

There are several new parameters that can be set via an inputs file:

- `amr_buf_width`: radius (in cells) of tagged cells in addition to those already tagged due to the criteria in `tag_boxes.f90`.
- `cluster_minwidth`: any newly created grids must be at least this many cells in each direction.
- `cluster_blocking_factor`: any newly created grids must have an integer multiple of this many cells in each direction.
- `cluster_min_eff`: This is a real number between 0 and 1 that controls how tightly the newly created grids match the tagged cells. As this value approaches 1, you will have more, smaller grids. Another way to think of this is that during the grid creation process, at least 100×`cluster_min_eff` percent of the cells in each grid at which the grid creation occurs must be tagged cells.
- `regrid_int`: frequency, in time steps, on when to regrid the simulation.

It is worth playing around with the inputs files to see what effect these parameters have on the grid structure.

3.4 Linear Solvers

The tutorial code `BoxLib/Tutorials/HeatEquation_EX5_F/` contains an implicit version of the heat equation example. Fortran90 `BoxLib`contains a "cell-centered" multigrid solver that solves linear systems of the form:

$$(\alpha\mathcal{I} - \nabla\cdot\beta\nabla)\phi = \text{RHS}, \tag{3.1}$$

where α, ϕ, and RHS are cell-centered `MultiFab` s, and β is an array of `MultiFab` s that are nodal in exactly one direction (i.e., one face-centered `MultiFab`for each spatial direction). The Laplacian-like term in the left-hand-side can be discretized in several ways. The simplest discretization option is similar to a 5-point (7-point in 3D) Laplacian:

$$\begin{aligned}\nabla\cdot\beta\nabla\phi_{ij} &= \frac{1}{\Delta x}\left[\beta_{i+1/2,j}\frac{\phi_{i+1,j}-\phi_{ij}}{\Delta x} - \beta_{i-1/2,j}\frac{\phi_{ij}-\phi_{i-1,j}}{\Delta x}\right] \\ &+\frac{1}{\Delta y}\left[\beta_{i,j+1/2}\frac{\phi_{i,j+1}-\phi_{ij}}{\Delta y} - \beta_{i,j-1/2}\frac{\phi_{ij}-\phi_{i,j-1}}{\Delta y}\right].\end{aligned} \tag{3.2}$$

A fully implicit discretization of the heat equation,

$$\frac{\phi^{n+1} - \phi^n}{\Delta t} = \left[\nabla \cdot (\nabla \phi)\right]^{n+1}, \tag{3.3}$$

is equivalent to

$$(\mathcal{I} - \Delta t \nabla \cdot \nabla)\phi^{n+1} = \phi^n. \tag{3.4}$$

Thus, we will set $\alpha = 1$, each $\beta = \Delta t$, and RHS $= \phi^n$.